METHOD, COMPUTER SYSTEM AND COMPUTER PROGRAM PRODUCT FOR DETERMINING THE EQUIVALENCE OF TWO BLOCKS OF ASSIGNMENT STATEMENTS

Background of the Invention

The present invention relates to the determination of the equivalence of two blocks of assignment statements, such as those used in computer programs. The invention provides a method and computer system for carrying out such a determination and also to a computer program product including a computer readable medium having recorded thereon a computer program for performing such determination.

An assignment statement assigns the value of an expression, the right-hand side of the statement, to a variable at the left-hand side of the statement. The two sides of the statement are connected by an assignment operator, which in many programming languages (including C and C++) is the equals sign (=). Thus, an example of an assignment statement in a programming language such as C might be x = x + 4. The interpretation of the assignment operator (=) in this statement is: take the current value of x (say, 3), add 4 to it and assign the result (7) to x as its new value. This operation is very different from the interpretation of the equal sign in algebra where the statement x = x + 4 would be meaningless since in algebra it is used to express an equation and not an assignment. In some programming languages the assignment operator is given the symbol (:=) to avoid confusion between the assignment and equation.

The present invention has for its object, a technique which is specifically relevant to the determination of whether or not two blocks of assignment statements are equivalent. By way of example, such a technique is useful in realising intermediate steps in program verification, program proving, and compiler initiated optimisation of source code. Expressed mathematically, the objective is as follows:

Given, (1) a set of input variables $\{x_1, x_2, \dots x_m\}$ and a set of output variables

$\{x_k, x_{k+1}, \ldots x_n\}$ with an overlap of variables in the two sets given by the set $\{x_k, x_{k+1}, \ldots x_m\}$ if $k \leq m$, and (2) two blocks of assignment statements B_1 and B_2 where B_1 comprises the set of assignments $\{{}^1S_1, {}^1S_2, \ldots {}^1S_M\}$ and B_2 comprises the set of assignments $\{{}^2S_1, {}^2S_2, \ldots {}^2S_N\}$, determine if the two blocks B_1 and B_2 provide equivalent computations for the output variables.

The term equivalent computation here means that the values for the output variables $\{x_k, x_{k+1}, \ldots, x_n\}$produced by B_1 are identical to those produced by B_2 given the input variables $\{x_1, x_2, \ldots, x_m\}$ even if the sets $\{{}^1S_1, {}^1S_2, \ldots {}^1S_M\}$ and $\{{}^2S_1, {}^2S_2, \ldots, {}^2S_N)$ are not identical. For example, given the input variables $\{x_1, x_2, x_3\}$ and output variables $\{x_2, x_3, x_4\}$, block B_1 might comprise the assignment statements:

$$
\begin{aligned}
&x_5 = x_1\\
&x_6 = 3*x_2\\
&x_3 = x_1 + 3*x_2\\
&x_1 = x_1 + x_2 + x_2\\
&x_2 = 2\\
&x_1 = x_1\\
&x_4 = x_5 + x_6
\end{aligned}
$$

while block B_2 comprises the statements:

$$
\begin{aligned}
&x_1 = x_1 + 2*x_2\\
&x_3 = x_1 + x_2\\
&x_4 = x_3\\
&x_2 = 2
\end{aligned}
$$

Inspection will reveal that, in either case, the output variables $\{x_2, x_3, x_4\}$ will produce the same computed values.

Summary of the Invention

The invention provides a computer implemented method for determining, in a computer environment, the equivalence, if any, of two blocks of assignment statements in a computer program, for use in compiler optimisation of source code, program verification, program proving, and like computing tasks, said method comprising the steps of:

(a) forming, for each block of assignment statements, a corresponding array, each array comprising a plurality of elements corresponding to respective ones of the statements and populating the elements with attributes of the statements including the expression at the right-hand side of the statement;

(b) processing, in each array, each assignment statement in turn, in the order from the last statement to the first, the processing comprising the inspection of each unprocessed assignment statement in turn, in the order from the last unprocessed assignment statement to the first, to determine if the variable appearing on the left-hand side of the unprocessed assignment statement appears on the right-hand side of the assignment statement being processed;

(c) during step (b), in each array, if the variable appearing on the left-hand side of the unprocessed assignment statement also appears on the right-hand side of the assignment statement being processed, replacing all occurrences of such variable on the right-hand side of the assignment statement being processed, non-recursively, by the right-hand side of the said unprocessed assignment statement;

(d) forming, from each array, a corresponding new block of assignment statements comprising the statements processed according to steps (b) and (c) less any statements which, after processing, is either an identity (the left and right sides of the statement are identical) or whose left-hand side variable is not one of the output variables;

(e) creating, from each new block of assignment statements, a corresponding new array, each array comprising a plurality of elements corresponding to respective ones of the statements and populating the elements with attributes of the statements including the expression at the right-hand side of the statement;

(f) sorting, in each new array, the array elements in alphabetical order using the output variable name as the key.

(g) comparing the arrays to detect the equivalence of two blocks of assignment statements.
The method, *inter alia*, successfully eliminates, from a block of assignment statements, all intermediate variables and statements which are identities and also those which are irrelevant to the computation of the output variables and brings the block to a form suitable for comparing two or more blocks of assignment statements.

The invention also provides an apparatus adapted to determine, in a computer environment, the equivalence, if any, of two blocks of assignment statements in a computer program, for use in compiler optimisation of source code, program verification, program proving, and like computing tasks, said apparatus comprising:

(a) forming means for forming, for each block of assignment statements, a corresponding array, each array comprising a plurality of elements corresponding to respective ones of the statements and populating the elements with attributes of the statements including the expression at the right-hand side of the statement;

(b) processing means for processing, in each array, each assignment statement in turn, in the order from the last statement to the first, the processing comprising the inspection of each unprocessed assignment statement in turn, in the order from the last unprocessed assignment statement to the first, to determine if the variable appearing on the left-hand side of the unprocessed assignment statement appears on the righthand side of the assignment statement being processed;

(c) during step (b), in each array, if the variable appearing on the left-hand side of the unprocessed assignment statement also appears on the right-hand side of the assignment statement being processed, replacement means for replacing all occurrences of such variable on the right-hand side of the assignment statement being processed, non-recursively, by the right-hand side of the said unprocessed assignment statement;

(d) forming means for forming, from each array, a corresponding new block of assignment statements comprising the statements processed according to steps (b) and (c) less any statements which, after processing, is either an identity (the left and right sides of the statement are identical) or whose left-hand side variable is not one of the output variables;

(e) creation means for creating, from each new block of assignment statements, a corresponding new array, each array comprising a plurality of elements corresponding to respective ones of the statements and populating the elements with attributes of the statements including the expression at the right-hand side of the statement;

(f) sorting means for sorting, in each new array, the array elements in alphabetical order using the output variable name as the key.

(g) comparison means for comparing the arrays to detect the equivalence of two blocks of assignment statements.

The invention further provides a computer program product including a computer readable medium having recorded thereon a computer program for determining, in a computer environment, the equivalence, if any, of two blocks of assignment statements in a computer program, for use in compiler optimisation of source code, program verification, program proving, and like computing tasks, said program comprising:

(a) forming process steps for forming, for each block of assignment statements, a corresponding array, each array comprising a plurality of elements corresponding to respective ones of the statements and populating the elements with attributes of the statements including the expression at the right-hand side of the statement;

(b) processing steps for processing, in each array, each assignment statement in turn, in the order from the last statement to the first, the processing comprising the inspection of each unprocessed assignment statement in turn, in the order from the last unprocessed assignment statement to the first, to determine if the variable appearing on the left-hand side of the unprocessed assignment statement appears on the righthand side of the assignment statement being processed;

(c) during step (b), in each array, if the variable appearing on the left-hand side of the unprocessed assignment statement also appears on the right-hand side of the assignment statement being processed, replacement steps for replacing all occurrences of such variable on the right-hand side of the assignment statement being processed, non-recursively, by the right-hand side of the said unprocessed assignment statement;

(d) forming process steps for forming, from each array, a corresponding new block of assignment statements comprising the statements processed according to steps (b) and (c) less any statements which, after processing, is either an identity (the left and right sides of the statement are identical) or whose left-hand side variable is not one of the output variables;

(e) creation process steps for creating, from each new block of assignment statements, a corresponding new array, each array comprising a plurality of elements corresponding to respective ones of the statements and populating the elements with attributes of the statements including the expression at the right-hand side of the statement;

(f) sorting process steps for sorting, in each new array, the array elements in alphabetical order using the output variable name as the key.

(g) comparison process steps for comparing the arrays to detect the equivalence of two blocks of assignment statements.

The present invention relates to determining if two syntactically correct blocks of assignment statements in a computer program are equivalent or not. Equivalence among more than two blocks of assignment statements may be determined by pair-wise comparison of the blocks.

Among its applications are compiler initiated optimisation of source code, where it is desirable to recognise multiple occurrences of blocks of assignment statements producing identical output so that such blocks can be evaluated once, and the result used in all the remaining instances.

Brief Description of the Drawings

For a better understanding of the invention, reference will be made, by way of example, to the accompanying drawings, in which:

Figure 1 is a simplified diagram of a computer system; and

Figures 2 and 3 are flow-charts for explaining the method of the invention.

Detailed Description

The invention includes a method for determining the equivalence of two blocks of assignment statements such as might be found in a computer program. The method may be implemented as a program for a computer, and the program may be stored on a storage medium, for example a CD-ROM, to form a program product according to the invention. Alternatively, a program product according to the invention may comprise a program made available for downloading from another computer. The computer program can be loaded into or made available to a suitable computer to form a computer system of the invention. Figure 1 shows one embodiment of such a computer system.

This embodiment comprises a so-called stand-alone computer, i.e. one that is not permanently linked to a network. It includes a display monitor 2, a keyboard 3, a microprocessor - based central processing unit 4, a hard-disc drive 5 and a random access memory 6 all coupled one to another by a connection bus. The keyboard 3 is operable for enabling the user to enter commands into the computer along with user data. As well as keyboard 3, the computer may comprise, a mouse or tracker ball (not shown) for entering user commands especially if the computer is controlled by an operating system with a graphical user interface.

To load program instructions into the memory 6 and/or store them on the disc drive 5 so that the computer begins to operate or to become operable in accordance with the present invention, the computer 1 comprises a CD-ROM drive 8 for receiving a CD-ROM 9.

The program instructions are stored on the CD-ROM 9 from which they are read by the drive 8. However, as will be well understood by those skilled in the art, the instructions as read by the drive 8 may not be usable directly from the CD-ROM 9. Instead, they may be loaded into the memory 6 and stored in the hard disc drive 5 and used by the computer 1 from there. Also, the instructions may need to be decompressed from the CD-ROM using appropriate decompression software on the CD-ROM or in the memory 6 and may, in any case, be received and stored by the computer 1 in a sequence different to that in which they are stored on the CD-ROM.

In addition to the CD-ROM drive 8, or instead of it any other suitable input means could be provided, for, example a floppy-disc drive or a tape drive or a wireless communication device, such as an infrared receiver (none of these devices being shown).

The computer 1 also comprises a telephone modem 10 through which the computer is able temporarily to link up to the Internet via telephone line 11, a modem 12 located at the premises of an Internet service provider (ISP), and the ISP's computer 13. Also connected to the Internet are many remote computers, such as the computer 14, from which information, software and other services are available for downloading to the computer 1. Furthermore, the computer 1 does not have to be in a stand-alone environment. Instead, it could form part of a network (not shown) along with other computers to which it is connected on a permanent basis. It could also be permanently coupled to or have a temporary link to an intranet. An intranet is a group of data holding sites similar to Internet sites and arranged in the same way as the Internet but accessible only to particular users, for example the employees of a particular company. Instead of modem 10, the computer 1 could have a digital hard-wired link to the ISP's computer 13 or the computer 1 could itself comprise a permanently connected Internet site whether or not acting as an ISP for other remote users. Instead of the invention being usable only through the local keyboard 3, it may be available to remote users working through a temporary or permanent link to computer 1 acting as ISP or simply as an Internet site.

Thus, instead of being provided by a local device such as the CD-ROM drive 8, the program instructions could be received or made available via the Internet from a remote computer such as the computer 14. Alternatively, the instructions could be received or made available from a computer connected with computer 1 via a network such as an intranet.

To carry out the method of the invention, the computer 1 is loaded with a suitable operating system, a compiler including a section for carrying out the method, or simply a specific utility for carrying out that method, and a section of source code which, it is assumed, contains the two sets of assignment statements of which the equivalence is to be determined. The compiler/utility is supplied to computer 1 from a portable storage medium such as a series of floppy discs (not shown) or a CD-ROM or from another computer such as the computer 14.

The method will now be described by reference to the three flow charts of Figures 2, 3 and 4. The numbered "steps" noted herein correspond to the step numbers of the flow charts.

Referring first to Figure 2, the two sets of assignment statements are identified in the relevant source code and then, to facilitate the analysis, each assignment statement is now reduced to a standardised format according to a predetermined set of rules as follows:

Step 1: Make variable names compliant

Initially, it is assumed that the statement is syntactically correct and does not contain any blanks. In the preferred embodiment, variable names appearing in the statement may comprise only lower-case alphabet characters, the underscore character, and digits. Also, a variable name may not start with a digit or end with an underscore. If these construction rules are not met, then the affected variable names are mapped (aliased) to alternative, but distinct, names obeying the construction rules, and these new names used instead. The right-hand side expression of the statement is then subjected to the following steps 2 to 12.

Step 2: Remove brackets

Brackets, if present, in the expression must be removed by carrying out the necessary operations needed to remove them, such as multiplying two parenthesised factors, discarding superfluous brackets, and so on.

Step 3: Insert leading operator if none present

The expression is put in the following form:

< unitary operator > < operand > < operator > < operand > ....
< operator > < operand >

where the unitary operator is either + (plus) or – (minus), and each operator is one of + (plus), - (minus), * (multiplication) or / (division). In the event that an expression does not commence with a unitary operator, one is added, i.e. a unitary operator + (plus) is inserted at the start of the expression. For example:

a * b/c becomes +a * b/c

Step 4: Map division by variable to multiplication by reciprocal of variable

Division by a variable, for example, the operator operand pair /x, is replaced by multiplication by the reciprocal of the variable, where the reciprocal of the variable is formed as a new variable by appending an underscore to the variable, i.e. /x is replaced by *x_ in the case of the given example.

<u>Step 5: Map division by constant to multiplication by reciprocal of constant</u>

Division by a constant, for example /5, is replaced by multiplication by the reciprocal of the constant, i.e., /5 is replaced by *.2 in the case of the given example.

<u>Step 6: Replace "+" and "-" by strings "+1*" and "-1*"</u>

Next all + (plus) operators are substituted with the string "+1*" so that "+" becomes "+1*". Similarly, all - (minus) operators are substituted with the string "-1*" so that "-" becomes "-1*". Thus, for example:

+x becomes +1*x

and

-x*y+z becomes –1*x*y+1*z

<u>Step 7: Convert constants to e-format</u>

Next the operands, which are constants (including the 1's introduced in the previous step) are converted into an e-format as follows:

" . [unsigned number]e[e-sign][unsigned exponent]"

where : [unsigned number] is an *n*-digit number comprising only digits and *n* is a predetermined fixed integer greater than 0; [e-sign] is the sign of the exponent and is one of > for plus or < for minus; and [unsigned exponent] is an *m*-digit number comprising only digits and *m* is a predetermined fixed integer greater than 0.

Thus, for example:

25 = $0.25*10^2$ becomes .250000e>02

and

0.025 = $0.25*10^{-1}$ becomes .250000e<01 where we have assumed n=6 and m=2. It is noted that any constant will be represented by a string of constant length m+n+3 characters in the e-format. Here e[e-sign] [unsigned exponent] represents the quantity 10 raised to the power [e-sign] [unsigned exponent], which must be multiplied to the number represented by . [unsigned number] to get the actual constant.

Now, the expression is free of the division operator and will contain at least one operand which is a constant. Each term in this expression will therefore have the following form:

< unitary operator > < operand > < * > <operand> .... <*> < operand >

where the unitary operator is either + (plus) or – (minus), and between two consecutive operands is the multiplication operator *. After the terms are identified, the [e-sign] of each constant is restored from < or > to – or + respectively.

Step 8: Sort operands in each term

In each term the operands are sorted (rearranged) in ascending order according to their ASCII value. This rearrangement is entirely permissible because the multiplication operator is commutative, i.e. the exchange of operands does not affect the result.

It is noted that no other variable will be able to place itself in the rearrangement between any particular variable and its reciprocal if they are both present. For example, if the variable "a" and its reciprocal "a_" are both present, they will be sorted so as to remain together as "*a*a_".

Step 9: Eliminate variable/reciprocal pairs

In the next step, all operator-operand sequences of the form "*a*a_" are eliminated from the term. For example, the expression a^3/a^2 will appear as "*a*a*a*a_*a_". After "*a*a_" has been eliminated from it, "*a*a*a_" will remain, from which "*a*a_" must, again, be eliminated. That is, the elimination process must be continued till no further elimination is possible.

Step 10: Consolidate constants in each term

After sorting, the operands which are constants will be bunched up at the beginning of the terms where they can be easily identified and replaced by a single constant. Thus, for example:

+.100000e+01*s*k*m*s_*.500000e+00

after arranging the operands in ascending order becomes

+.100000e+01*.500000e+00*k*m*s*s_

and after eliminating the units s and s_ and consolidating the constants, the term becomes

+.500000e+00*k*m

At this stage a term will have the following form:

< unitary operator > < constant > < * > < operand > ... < * > < operand >

where each operand is a variable name, whose ASCII value is not lower than that of its preceding operand, if any. This is the reduced form of a term. In the reduced form, the non-constant part of a term is called a variable-group. For example, if the term in the reduced form is "+.250000e+01*m*m*s", then its variable-group is "*m*m*s".

Step 11: Consolidate like terms

In an expression, all those terms whose variable-groups match, are combined by modifying the constant in one of the terms, and eliminating the others.

Step 12: Sort terms in the expression

Finally, the reduced terms in the expression are rearranged in an ascending order according to the ASCII value of their respective variable-groups. In this final form, the expression is said to be in its reduced form. Note, in particular, that no two terms in a reduced expression will have the same variable-group.

Having obtained the reduced right-hand side expressions, the method continues by carrying out certain array processing operations as will be described. For convenience, conventions and data structures similar to those used in the C/C++ programming language have been used in the following part of the description. However, the invention is not specific to the C/C++ programming languages.

To extract certain attributes of an assignment statement *S*, expressed as *v* = *e*, there is defined a data structure *D* with elements *v, e, p, d, u, nVars, tag*. Here *v* is the variable appearing on the left-hand side of the statement; *e* is the right-hand side expression; *p* is the list of the variables appearing in *e*; *d* is the sublist of variables from *p*, which have appeared on the left-hand side of an earlier assignment statement; *u* is the list of, as yet, undefined variables in *e*; *nVars* is the number of variables (defined or undefined) appearing in *p*, and *tag* has the value –1 if the assignment is an identity, 0 if *e* is a constant value, 1 if *v* does not appear in *p*, 2 if *v* appears in *p*. In the programming language C/C++, the data structure could be expressed as:

```
struct _EQ_ATTRIBS{
  char *v;          //lhs variable
  char *e;          //rhs expression
  char *p;          //list of rhs variables in equation
  char *d;          //list of rhs variables defined by one or more previous
                    //assignments
  char *u;          //list of undefined variables in rhs expression
  int nVars;        //number of variables in p
  int tag;          //equation tag; -1 (identity), 0 (rhs = const), 1 (non-recursive),
                    //2(recursive)
} D;
```

Further, there is created from a given block of assignment statements B, an array of the structure _EQ_ATTRIBS, D, expressed in a standard form. A property of this standard form is that, for a given input variables set, if two blocks of assignment statements B_1 and B_2 produce identical expressions for corresponding output variables for the output variables set, then they will also produce a common D.

Referring now to Figure 3 in turn, the array processing operations comprise the following steps:

Step 13. If M is the total number of statements in B, create the empty array D of dimension M. Let the i-th element of D be given by $D[i] \equiv (v_i, e_i, p_i, d_i, u_i, nVars_i, tag_i)$.

Step 14. For every S_i in B, determine its attributes. Let $D[i]$ contain the attributes of S_i. Note that a variable is undefined in S_i if it does not appear in either the input variables set or as a left-hand side variable in any previous assignment statement. All such undefined variables are placed in the element u_i of $D[i]$. Then arrange the variables in the lists p_i and d_i in ascending order according to the ASCII value of the variables, placing only a comma between variables to separate them from each other. This is done to facilitate comparisons between similar entities by means of string comparison.

Step 15. For every S_i in B, determine if it is an identity (its *tag* value will be –1). If it is an identity, mark the statement for elimination from B (say, by emptying the contents of $D[i]$).

Step 16. For every S_i in B, check if it has one or more undefined variables.(i.e. its u_i is non-empty). If such a statement is found, terminate the algorithm with the message that the block of statements B has one or more undefined variables along with the list of the undefined variables collected in u_i, for each of the statements in B.

Step 17. Begin with the last statement S_M in B and move up to the first statement S_1. For every S_i, check if its d_i is non-empty. If it is non-empty define an index j. Begin with $j = i - 1$ (with j > 1) and go backwards to $j = 1$. For each v_j appearing in S_i, replace v_j by its corresponding e_j. Note that this is a string replacement and to carry it out with semantic correctness e_j must be enclosed within the parentheses "()". That is, replace the string, which represents v_j with the string "(e_j)" where e_j is the string representation of the right-hand side of S_j. Further, after the replacement, ensure that if v_j reappears in "(e_j)", it is left untouched (since it would belong to the left-hand side of a still earlier assignment statement). This is easily done, for example, by scanning from left to right the expression e_i and after every replacement of v_j moving the pointer to the right of ')' of "(e_j)". This ensures that any v_j appearing in the assignment now will be correctly replaced by its corresponding e_j as we go up the index j towards 1. After operation with $j = 1$ is completed, mark S_i for deletion if it reduces to an identity after the replacements are completed, otherwise empty out p_i, d_i, u_i, $nVars_i$, tag_i (if not already empty) since they are no longer relevant. Step 17 is a crucial step, and will need to be coded carefully. The right-hand sides of the statements remaining will contain only the input variables and those output variables, which have appeared on the left-hand side of a previous assignment statement.

<u>Step 18.</u> Renumber the statements in B, skipping all statements marked for deletion but otherwise maintaining their sequence. Let these be S_1 to S_r. (Note: $r \neq M$, if identities were found in steps 15 and 17. The number of identities found will be $M - r$). Define a *current output variables list* and copy the variables in the output variables set into it. Begin with $j = r$ and move backwards to $j = 1$. For every S_j, check if its v_j is in the current output variables list. If it is, retain the statement in B and delete the variable v_j from the current output variables list. If it is not, mark the statement for deletion from B by emptying out its entries in $D[j]$. Note that at the end of this step all intermediate variables (i.e. variables, which are neither input variables nor output variables) would have been eliminated from B. So would also statements which do not contribute to the computation of the output variables set.

<u>Step 19.</u> If, at the end of step 18, the current output variables list is not empty, terminate the algorithm with the message that some output variables have not been calculated and list those variables (these will be the variables remaining in the current output variables list). Otherwise move to step 20.

Step 20. At this stage, the number of assignment statements left in B will be equal to the number of variables $(n - k + 1)$ in the output variables set. Destroy B. Recreate the assignment statements of a new B, in the sequence they appear in **D** (skipping those which were marked for deletion in step 18), by concatenating the strings v, e, and the character '=' in the form 'v=e'. Destroy **D**. Construct a new **D** for the $(n - k + 1)$ assignment statements.

Step 21. Rearrange the $(n - k + 1)$ data structures in **D** in alphabetical order using v as the key. This is the final standard form of **D** for the original set of the assignment statements.

If two blocks of assignment statements B_1 and B_2 each produces the same **D**, then their output variables will obviously produce the same values of execution.

The algorithm as described above eliminates, from a block of assignment statements, all intermediate variables and statements which are identities and also those *which are irrelevant to the computation of the output variables*. It retains only those statements, which assign final values to the output variables. Since the attributes of each of the statements (contents and it's hierarchical position in B) is preserved in the array **D**, each element $D[i]$ is self-contained and can be read in any order. Therefore **D** can be sorted as a list and put into a desired standard form.

As an example of the operation of the algorithm, the first block of assignment statements mentioned in the background section of this specification are used, namely:

```
x5 = x1
x6 = 3*x2
x3 = x1 + 3*x2
x1 = x1 + x2 + x2
x2 = 2
x1 = x1
x4 = x5 + x6
```

To this, the steps of the algorithms are applied as shown below:

Each statement is reduced according to steps 1 to 12. For illustration, one statement, namely x3 = x1 + 3*x2, will be reduced.

Step 1

To make variables compliant, they are mapped to x3, x1 and x2 respectively. The right-hand side of the statement, ie x1 + 3*x2 is then subject to rules 2 to 12.

Step 2

There are no brackets.

Step 3

A leading operator is added to give +x1+3*x2.

Steps 4 and 5

There are no division operators.

Step 6

Plus and minus operators are replaced by "+1*" and "-*1" to give +1*x1+1*3*x2.

Step 7

Put constants in e-format

+.100000e>01*x1+.100000e>01*.300000e>01*x2

Thus, the expression has two terms +.100000e>01*x1 and +.100000e>01*.300000e>01*x2. Having identified the two terms, the e-signs of the constants are restored to give +.100000e+01*x1 and +.100000e+01*.300000e+01*x2.

Steps 8 to 10

The operands are already sorted in each term and there are no variable/reciprocal pairs. However, the constant operands of the second term can be consolidated to give

+.300000e+01*x2.

Steps 11 and 12

The variable groups of the respective terms do not match. In fact, each variable group contains just one variable, namely of x1 and x2 respectively. Thus, the reduced form of the statement is:-

x3 = +.100000e+01*x1+.300000e+01*x2.

Steps 13-15

Note that *M* = 7. At the end of step 15 the array D has the following entries

D[1] = {"x5", "+.100000e+01*x1", "x1", "", "", 1, 1}
D[2] = {"x6", "+.300000e+01*x2", "x2", "", "", 1, 1}
D[3] = {"x3", "+.100000e+01*x1+.300000e+01*x2", "x1,x2", "", "", 2, 1}
D[4] = {"x1", "+.100000e+01*x1+.200000e+01*x2", "x1,x2", "", "", 2, 1}
D[5] = {"x2", "+.200000e+01", "", "", "", 0, 0}
D[6] = {"x1", "+.100000e+01*x1", "x1", "x1", "", 1, -1} (see note)
D[7] = {"x4", "+.100000e+01*x5+.100000e+01*x6", "x5,x6", "x5,x6", "", 2, 1}

Note that the contents of *D*[6] just prior to marking it for elimination are shown here for illustration. Since its *tag* = -1, it refers to an identity statement, which must be eliminated in a later step.

Step 16

No action since no undefined variables were found.

Step 17

At the end of step 17, the array D has the following entries:

D[1] = {"x5", "+.100000e+01*x1", "", "", "", ,}
D[2] = {"x6", "+.300000e+01*x2", "", "", "", ,}
D[3] = {"x3", "+.100000e+01*x1+.300000e+01*x2", "", "", "", ,}
D[4] = {"x1", "+.100000e+01*x1+.200000e+01*x2", "", "", "", ,}
D[5] = {"x2", "+.200000e+01", "", "", "", , }
D[6] = {"", "", "", "", "", , }
D[7] = {"x4", "+.100000e+01*x1+.300000e+01*x2", "", "", "", , }

Notice that the intermediate variables x5, and x6 have been eliminated from the right-hand sides of the statements. No new identities were found at the end of this step, and r = 6.

Step 18

The current output variables list at the beginning of step 18 is "x2, x3, x4". At the end of step 18, the array D has the following entries:

D[1] = {"", "", "", "", "", , }
D[2] = {"", "", "", "", "", , }
D[3] = {"x3", "+.100000e+01*x1+.300000e+01*x2", "", "", "", , }
D[4] = {"", "", "", "", "", , }
D[5] = {"x2", "+.200000e+01", "", "", "", , }
D[6] = {"x4", "+.100000e+01*x1+.300000e+01*x2", "", "", "", , }

and the current output variables list is empty. Notice that only the output variables x2, x3, x4 appear on the left-hand side of the statements (the v_i element of $D[i]$), and the right-hand sides (the e_i element of $D[i]$) contain only defined input and output variables.

Step 19

The current output variables list is found to be empty. Move to step 20.

Step 20

The number of statements left in *B* are 3 (same as the number of output variables) corresponding to x2, x3, x4. Reconstruct them to produce the new *B*.

S[1] = "x3=+.100000e+01*x1+.300000e+01*x2"
S[2] = "x2=+.200000e+01"
S[3] = "x4=+.100000e+01*x1+.300000e+01*x2"

The new **D** is given by

D[1] = {"x3", "+.100000e+01*x1+.300000e+01*x2", "x1,x2", "x2", "", 2, 1 }
D[2] = {"x2", "+.200000e+01", "", "", "", 0, 0 }
D[3] = {"x4", "+.100000e+01*x1+.300000e+01*x2", "x1,x2", "x2", "", 2, 1 }

Step 21

The rearranged (sorted on *v*) **D** is given by

D[1] = {"x2", "+.200000e+01", "", "", "", 0, 0 }
D[2] = {"x3", "+.100000e+01*x1+.300000e+01*x2", "x1,x2", "x2", "", 2, 1 }
D[3] = {"x4", "+.100000e+01"x1+.300000e+01*x2", "x1,x2", "x2", "", 2, 1]

Step 22

After repeating steps 1 to 21 for the second block of statements, namely,

$x_1 = x_1 + 2^* x_2$

$x_3 = x_1 + x_2$

$x_4 = x_3$

$x_2 = 2$

A final **D** for the second block is found to be

```
D[1] = {"x2", "+.200000e+01", "", "", "", 0, 0 }
D[2] = {"x3", "+.100000e+01*x1+.300000e+01*x2", "x1,x2", "x2", "", 2, 1 }
D[3] = {"x4", "+.100000e+01*x1+.300000e+01*x2", "x1,x2", "x2", "", 2, 1 }
```

The two final arrays D are compared and, as may be seen are identical.

The algorithm described here provides a means for determining if two blocks of assignment statements are equivalent or not. The applications of such an algorithm are, among others, in program verification, program proving, and compiler initiated optimization of source code.

It will be emphasised that the invention is not limited to the specific embodiment described above but includes modifications and developments within the purview of those skilled in the art and limited only by the following claims.